NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D2.1
One-sided Matrix Factorizations

April 2017

Document information

Scheduled delivery	2017-04-31
Actual delivery	(not yet delivered)
Version	0.1
Responsible partner	UNIMAN

Dissemination level

CO — Confidential

Revision history

Date	Editor	Status	Ver.	Changes
2017-03-27	Samuel Relton	Draft	0.1	Initial version of document produced.

Author(s)

Negin Baghepour (UNIMAN)
Samuel Relton (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

First Last (UMU/UNIMAN/STFC/INRIA)
First Last (UMU/UNIMAN/STFC/INRIA)

Contributors

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

List of Figures

List of Tables

1 Introduction

The primary use of high-performance linear algebra software in scientific applications is in solving linear systems. Whether one needs to solve a system arising from the discretization of a partial differential equation, fit a model to some data in a least-squares sense, or solve a KKT system in an optimization problem, there will undoubtedly be linear systems to solve. Since the advent of "big data" these systems have become larger, necessitating the use of high-performance linear algebra routines to solve them in a reasonable amount of time. Usually such dense systems are solved using either the Cholesky, LU, QR, or symmetric-indefinite factorizations (collectively known as the one-sided factorizations).

In this work we aim to compare different implementations of these routines using the novel task-based programming paradigm. Whilst this has existed for a number of years, it has only recently appeared in OpenMP 4.0. Overall, our efforts in applying OpenMP to this problem are collated in the new release of the PLASMA library [1] which aims to implement the entire of the LAPACK standard [3] in this fashion.

The task-based programming paradigm breaks each linear algebra problem into a series of tasks operating on blocks of data. These tasks can depend on one another: for instance in a typical panel factorization algorithm the tasks used to factorize the panel need to be completed before the tasks to update the trailing matrix can begin. Typically these interdependencies are expressed using a directed acylic graph (DAG), which is also known as a task-graph. As an example, the DAG for a small Cholesky factorization is shown in Figure 1 below.

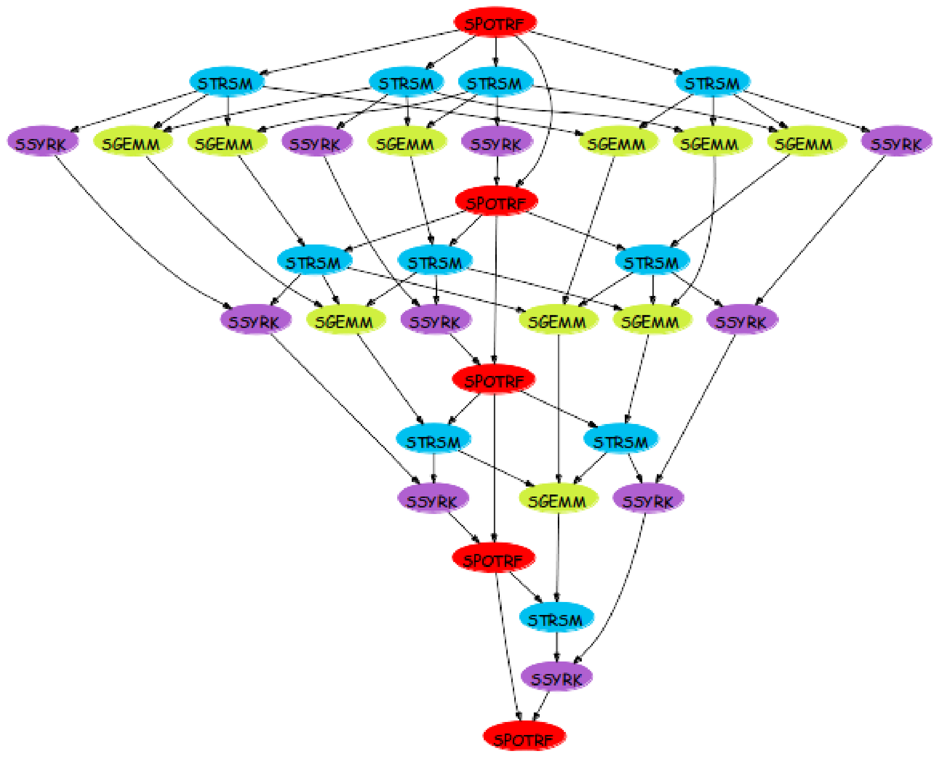


Figure 1: DAG for a small Cholesky factorization.

The red, blue, purple, and green tasks correspond to spotrf, strsm, ssyrk, and sgemm tasks whilst the arrows show the dependency between tasks. Each task will (usually) be performed by a single core and each completed task leads to further tasks becoming available. Once the dependencies for a task have been fulfilled it can immediately be assigned to a core. Some more detail on this model of programming used in a linear algebra context is given in section 2.

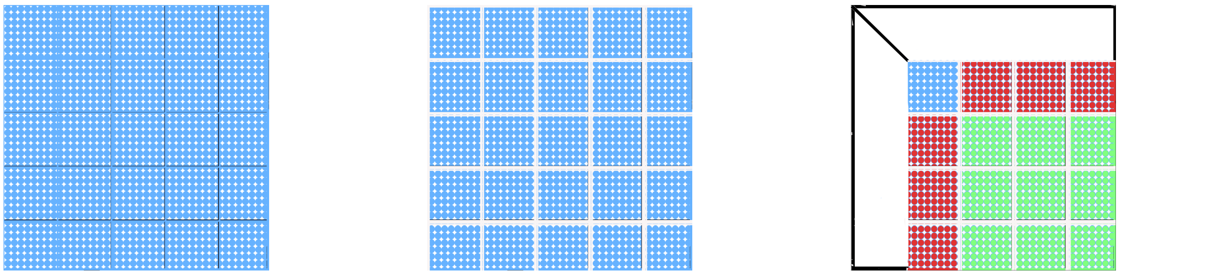
The goal of this report is to compare implementations of the four factorizations mentioned above using a variety of runtime systems in a multi-core environment. Each system is utilizing the same DAG and differs only in the way that tasks are assigned to the available cores. We also compare against reference Intel MKL for completeness. The performance of each implementation is measured on modern computer architectures: an Intel Broadwell NUMA node and the Intel Xeon Phi (codenamed Knights Landing).

The rest of this document is arranged as follows. Section 2 gives an introduction to the use of tile-based algorithms and task-based programming in linear algebra. In section 3 we give a brief summary of the capabilities of the various runtime systems used to implement the four factorizations. Section 4 gives more detail on the software libraries and architectures used in our experiments. Sections 5–8 then contain the experimental results for the four one-sided factorizations. Finally we summarize the results and give some conclusions in section 9.

2 Tile layout one-sided factorization

While LAPACK [4] linear algebra factorization algorithms have been successful in exploiting the first cache-based architectures, they have shown significant limitations on modern many/multi-core architectures and many well-tuned LAPACK-style kernels fail to achieve a satisfactory performance on these architectures [2]. As reported by Dongarra et al. in [7], three factors contribute to this performance penalty. The first factor is the fork-join parallelism model adopted in LAPACK. This model induces a high overhead on massively parallel architectures since it introduces many unnecessary synchronization points, keeping many computation resources frequently idle. Secondly LAPACK processing at coarse-grained granularity, block-column level (also known as panel), fails to exhibit enough parallelism to keep all the cores busy. Finally while all factorization algorithms in LAPACK are based on a recursive panel update followed by the corresponding trailing matrix update, the panel update uses memory-bound operations.

In order to use modern many/multi-core architectures at full efficiency, a new generation of linear algebra libraries such as PLASMA [6] and FLAME [8] have cast LAPACK block-column algorithms into tile algorithms. Tile algorithms enjoy the property of addressing each of the three drawbacks that keep LAPACK from providing a reasonable performance on modern massively parallel architectures. In fact, tile algorithms operates at a finer granularity by dividing the whole matrix into small square tiles which are more likely to fit into the L2 cache of a CPU as illustrated in Figure 2.


(a) Initial matrix.

(b) 5 × 5 tile matrix.

(c) Many kernels operating on different tiles in parallel.

Figure 2: Illustration of matrix division in square tiles as the case in tile algorithms. This helps working at a finer granularity to keep the maximum of the cores busy.

The order of execution of the tasks in tile algorithms are commonly represented in form of a DAG where each node represents a task, while the edges represent the data dependencies between the tasks. These tasks are then scheduled thanks to a runtime system which check the dependencies and takes care of launching tasks on appropriate cores.

The superiority of the tile layout algorithms over traditional approaches has been elegantly demonstrated through one-sided factorization benchmark suite reported in [2].

2.1 Tile QR factorization

2.2 Tile LU factorization

2.3 Tile Cholesky factorization

2.4 Tile LDLT factorization

3 Runtime systems

Each runtime system used in this report has a variety of features that makes them unique. In this section we will describe the unique features of each runtime system that we hope to compare. Even though this report focuses on multi-core environments, some of the runtime systems also support the use of accelerators and can scale to distributed architectures with minimal effort. The runtime systems that we will be using are

- OpenMP,
- Quark, and
- StarPU.

First, OpenMP is the standard way to parallelize code over a multi-core architecture. The pragma `#pragma omp parallel for` is a well-known method for parallelizing loops over the available cores. However, it is only recently that OpenMP began to support task-based programming. This is the main drawback of using task-based OpenMP currently: advanced features, such as assigning a priority to critical tasks or choosing from a variety of different task scheduling strategies, are either not in the current OpenMP standard or have not been widely implemented. This is due to the relative immaturity of task-based programming within OpenMP. Meanwhile, the benefits to using OpenMP are its wide availability and lightweight framework (meaning there are few significant overheads when using the task-based paradigm). As is well known, currently OpenMP does not widely support the use of accelerators and is not designed for distributed memory computation.

Second, Quark is a research project implementing task-based programming from ICL at the University of Tennessee[1]. It was one of the first frameworks to support this style of programming and as such does not follow all of the standards defined by the OpenMP Standardisation Committee. Since Quark is a fairly old research project it is not under current development: at the moment it is still highly relevant but this will of course fade in the coming years as OpenMP evolves. Quark does not support accelerators or distributed memory.

Finally, StarPU is a runtime system built by Inria Bordeaux [5]. The system supports the use of both GPUs and distributed memory computation, though neither of these features are used in this report to ensure a fair comparison. Another key advantage of StarPU over the other runtime systems is the incorporation of multiple different task scheduling strategies. In this report we will use the "eager", "dmda", and "ws" strategies.

First, in the "eager" strategy, each core draws tasks independently from a centralised queue whenever they become idle. This strategy does not allow for data prefetching since the scheduling decision is taken as late as possible. Meanwhile, the "dmda" (deque model data aware) strategy uses an estimate of the runtime (and memory transfer time) of each task to schedule them into multiple queues for each core, aiming to minimize the overall runtime. The downside to this strategy is the more complicated scheduling required, which may lead to overheads in task scheduling; these overheads would be insignificant on larger distributed computations where data transfer times dominate, though it remains to be seen how it performs on a single node. Each run of the computation using the "dmda" strategy gives the scheduling system more information upon which to base it's future decisions. Finally, the "ws" (work-stealing) strategy allows each core to steal work from

[1]More information available at `http://icl.utk.edu/quark/` as of 23rd March 2017.

Table 1: Architecture details for the Intel Broadwell NUMA node and the Intel Xeon Phi (KNL).

Platform	Xeon E5-2690 v4	Xeon Phi 7250
Cores	2×14 (2.6GHz)	68 (1.4GHz)
On-chip Memory	L1 32KB (per core)	L1 32KB
	L2 256KB (per core)	L2 34MB
	L3 35MB (per NUMA island)	MCDRAM 16GB
Main Memory	128GB DDR4	320GB DDR4
Bandwidth	76.8 GB/s	115.2 GB/s
Compiler	gcc 5.4.0	gcc 6.1.0
MKL version	17.0.1	17.0.2

the other cores when they becomes idle. This is designed to keep all cores busy at all times.

There are a number of other scheduling strategies implemented within StarPU but these are the ones best-suited to operation on a single node. The downside to StarPU is the additional overhead in task scheduling as a result of the numerous advanced features supported. Therefore, the support for accelerators and distributed memory means that StarPU is readily applicable to heterogeneous computing environments. We also make use of the KStar source-to-source compiler in this work, which automatically converts OpenMP task-based programs to StarPU programs[2].

There are also other runtime systems that could be considered, in particular ParSec (also from ICL at the University of Tennessee)[3]. ParSec supports the use of accelerators and distributed memory machines, but is currently rather difficult to use. The ParSec team is currently working on a simplified interface, similar to that used by StarPU, along with a similar source-to-source compiler for converting OpenMP code.

4 Experimental setup

Each of our experiments covering the four one-sided factorizations will be performed on two different architectures, to show the level of performance that can be expected on modern multi-core platforms. In Table 1 we describe each architecture in more detail and list the compilers and versions of MKL used but, briefly, we have:

- a 2-socket NUMA node with Intel Broadwell processors, and
- the new Intel Xeon Phi (codenamed Knights Landing, i.e. KNL).

In each scenario we will compare a number of different software libraries for the factorizations which, in each case, will be linked with Intel MKL BLAS. Therefore, the software libraries differ only in their implementation of the routines at the LAPACK level of abstraction, along with the scheduling system used in the task-based libraries. The libraries used are:

- Intel MKL,

[2]Downloaded from `http://kstar.gforge.inria.fr/#!index.md` on 23rd March 2017.
[3]Available from `https://bitbucket.org/icldistcomp/parsec` as of 23rd March 2017.

- PLASMA 2.8.0 (Quark),
- PLASMA 17 (OpenMP), and
- PLASMA KStar (StarPU).

First, we use the vendor optimized versions of LAPACK to compute each operation. The second and third libraries are different versions of the PLASMA library, one of which uses a runtime called Quark whilst the other uses OpenMP to schedule the tasks. Both libraries use a task-based programming model and involve splitting large matrices into "tiles", upon which sequential BLAS operations occur. Computing multiple tasks simultaneously is the major source of parallelism within these two libraries. Finally, we have used the KStar source-to-source compiler to automatically convert the OpenMP version of PLASMA to use StarPU. This allows us to use the advanced scheduling strategies implemented in StarPU: we use the "eager", "dmda',' and "ws" strategies in our experiments as discussed in the previous section. Due to some issues with the automatic conversion of the source code, we are unable to use PLASMA KStar for all routines: we can use PLASMA KStar only for the Cholesky and QR factorizations.

For each of the one-sided factorizations, and on each architecture, we will test the performance of each implementation as the matrix size increases: typically performance increases with the size of the matrix until some maximal data-throughput rate is reached.

At this point, it is worth reiterating that the goal of this report is to compare the runtime systems themselves and not the actual performance obtained. We are using *untuned* versions of PLASMA in these experiments, meaning that much better performance can be obtained after autotuning, which is the subject of NLAFET deliverable D6.4. Therefore comparisons with MKL at this point may be considered rather premature, though it is important to compare these preliminary implementations with state-of-the-art vendor releases.

5 Cholesky factorization

Figure 3 gives the performance results when computing a Cholesky factorization on the NUMA node in each of the four standard precisions. In all four precisions we see that StarPU with the "eager" strategy gives the best performance over essentially all matrix sizes. This is closely followed by the "ws" strategy and Quark. In double complex precision the "eager" strategy gives the best performance until matrices with $n > 13000$ are considered, after which MKL and OpenMP perform the factorization faster.

In Figure 4 we perform the same experiment on the KNL. As is clear from the graphs of all four precisions, MKL and OpenMP are making very efficient use of all 68 cores in the machine: the performance of the other PLASMA-based implementations begins to stagnate whilst MKL and OpenMP keep increasing. With further autotuning described in NLAFET deliverable D6.4 we will be able to increase the performance substantially. In all cases we see that Quark and StarPU with the "eager" scheduler are the best amongst the PLASMA implementations, and the best overall until the largest test matrices are used. OpenMP appears to be lagging behind the other implementations in performance most of the time, but does not suffer the same stagnation as Quark and StarPU.

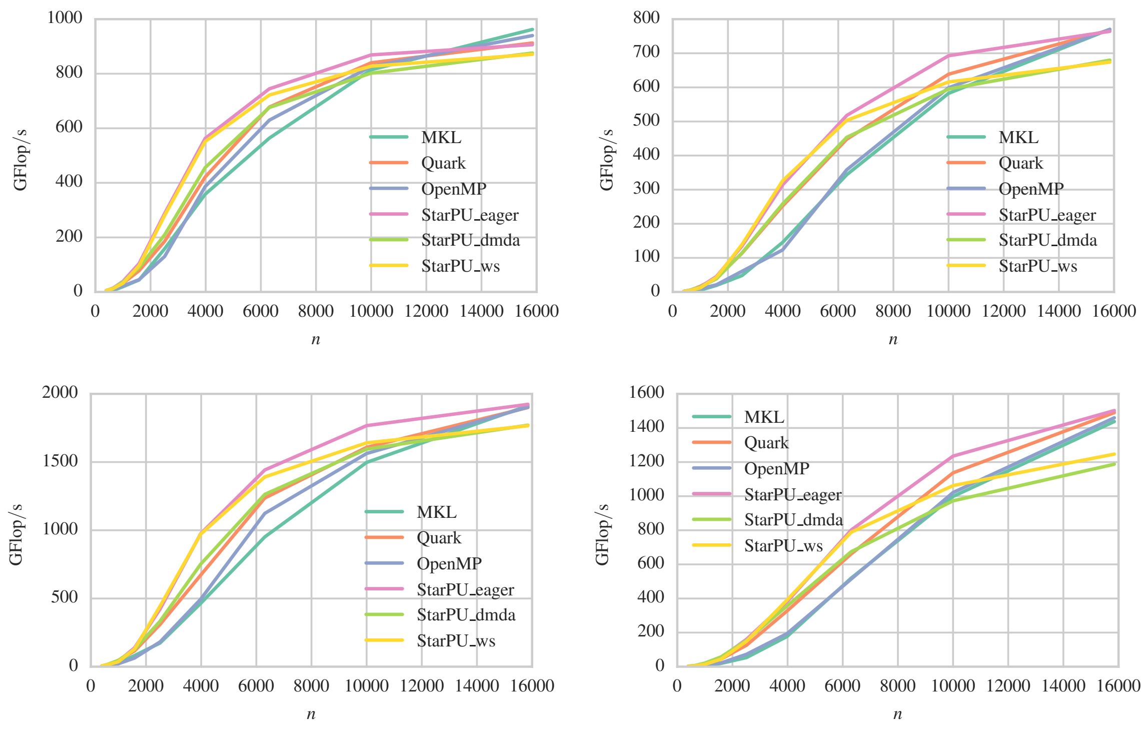


Figure 3: Performance of Cholesky factorization on NUMA node. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

6 *LU* factorization

Next we look at the *LU* factorization. The KStar source-to-source compiler was unable to convert the OpenMP implementation into StarPU here, so the various StarPU versions do not appear in this section.

In Figure 5 we plot the results from the NUMA node. Here we see that MKL and Quark give relatively similar performance, though Quark is significantly slower for the larger matrices in our experiments. In all cases OpenMP lags behind all other implementations: this is due to the way that task-dependencies have been expressed in the new PLASMA implementation, leading to less parallelism being expressed, and should not be considered a deficiency of the runtime itself. The PLASMA development team are currently exploring options to rectify the situation which will be released imminently.

In Figure 6 we perform the same experiment on the KNL system. As before, the MKL implementation scales well whilst the performance of Quark stagnates and even decreases slightly for very large matrices. Meanwhile, the lower level of parallelism expressed by the OpenMP implementation leads to an enormous performance hit on the KNL: the large number of cores need lots of parallelism to be utilised efficiently. When the OpenMP version is reimplemented we expect to see performance similar to that of MKL. One interesting feature of these plots is that for the smaller matrices in our tests, Quark is significantly faster than both MKL and LAPACK.

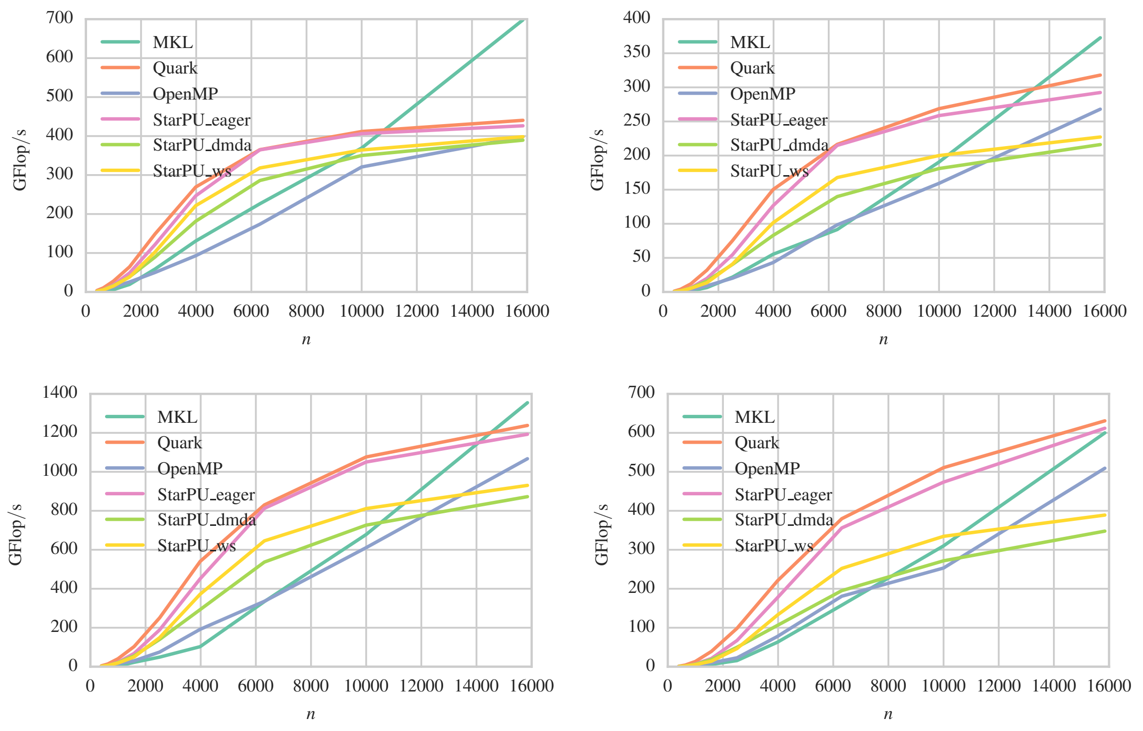


Figure 4: Performance of Cholesky factorization on the KNL. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

7 QR factorization

The third factorization we consider is QR. The KStar source-to-source compiler was able to generate StarPU code for this particular routine so all runtimes are present.

In Figure 7 we have the performance of the different implementations on the NUMA node for all four precisions. For double and double complex precision on the top row, we see that initially StarPU with the "eager" and "ws" strategies perform best, closely followed by Quark. However, after matrices of size 8000–10000 (depending upon the precision) are considered, MKL takes the lead whilst the performance of the other runtimes begin to stagnate. OpenMP is often the worst performer in these cases, and also stagnates when larger matrices are used. For complex and single precision the behaviour is rather similar, though StarPU with the "eager" scheduler is competitive until matrices of size larger than 10000 are used.

We perform the same experiment on the KNL in Figure 8. Here we see that all runtimes except for MKL stagnate very quickly and give relatively poor performance by comparison. These issues may well be resolved by utilising the autotuning described in NLAFET deliverable D6.4. Even though the performance leaves much to be desired, we can see that StarPU with the "eager" strategy and Quark perform significantly better than the other PLASMA-based implementations.

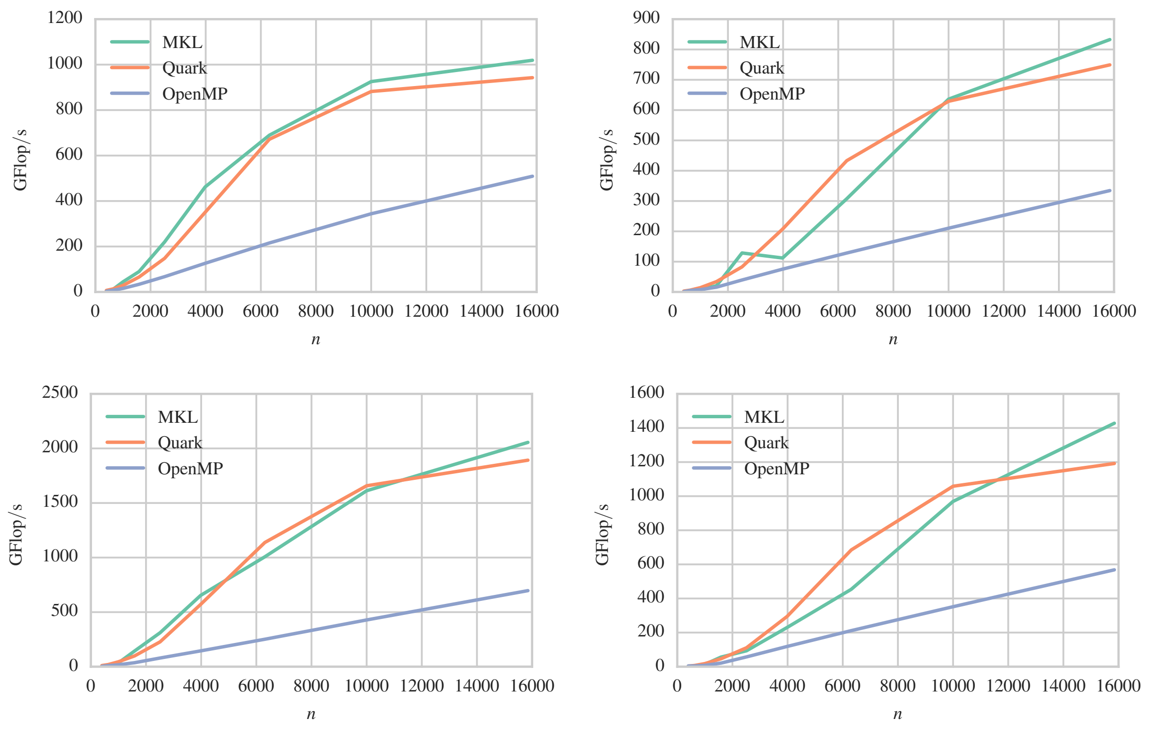


Figure 5: Performance of *LU* factorization on NUMA node. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

8 Symmetric-indefinite factorization

Finally, we look at the symmetric-indefinite factorization. Unfortunately, KStar could not generate a StarPU version of this function so we can only compare OpenMP and Quark MKL in this scenario.

The results obtained on the NUMA node are given in Figure 9. We see that clearly MKL is performing very well, but OpenMP is significantly better than Quark. Interestingly, none of the implementations are stagnating when the larger matrices are used.

The corresponding results on the KNL are shown in Figure 10. The behaviour is very similar to the previous experiment although Quark appears to be stagnating in some of these experiments.

9 Conclusions

It is clear from our experiments that our current implementations in Quark, OpenMP, and StarPU are not optimal. In order to catch up with MKL for larger matrices a significant amount of autotuning will need to be performed (see deliverable D6.4). However, the main goal of this report was to compare the various runtime systems.

It is clear that, when KStar was able to generate a StarPU version of our code, that StarPU with either the “eager” or “ws” strategies clearly outperformed OpenMP and Quark. Since StarPU also supports GPUs and distributed memory computation, it becomes the clear winner in our runtime comparison.

In further work we hope to compare against PaRSEC, once their source-to-source

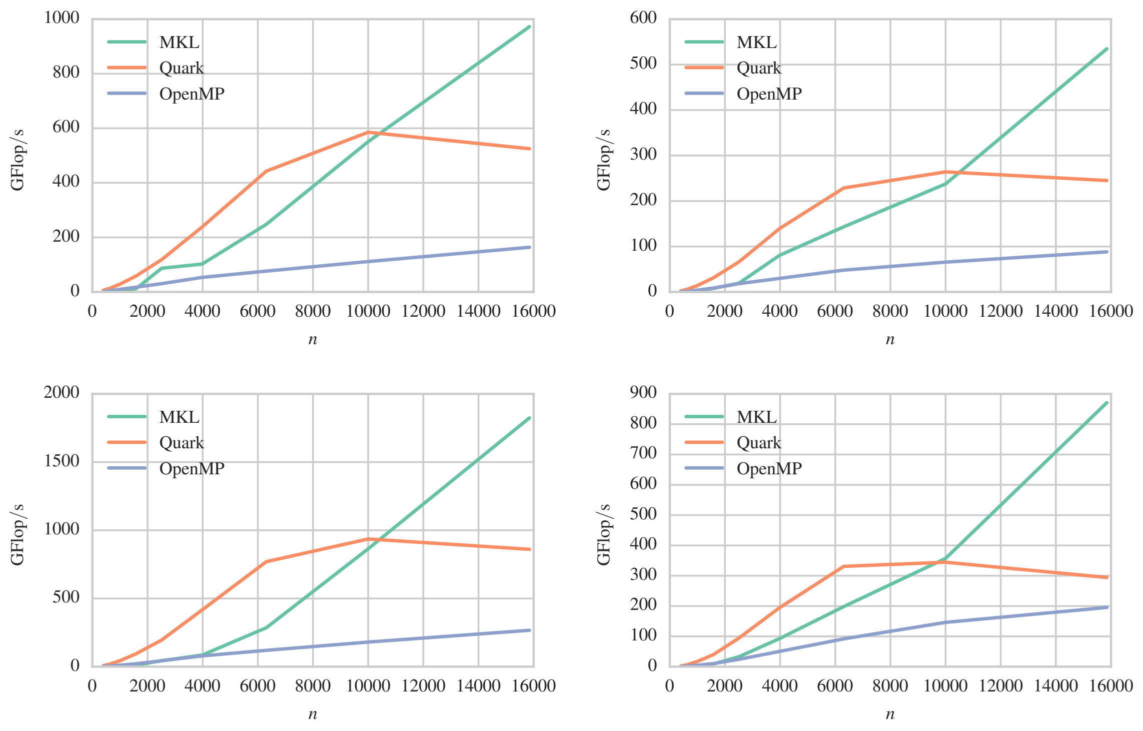


Figure 6: Performance of *LU* factorization on the KNL. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

compiler is completed. We would also like to recompare the PLASMA-based implementations against LAPACK and MKL once the autotuning from deliverable D6.4 has been incorporated.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[2] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarrra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 20. ACM, 2009.

[3] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide.* Third edition, 1999.

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide.* SIAM, 1999.

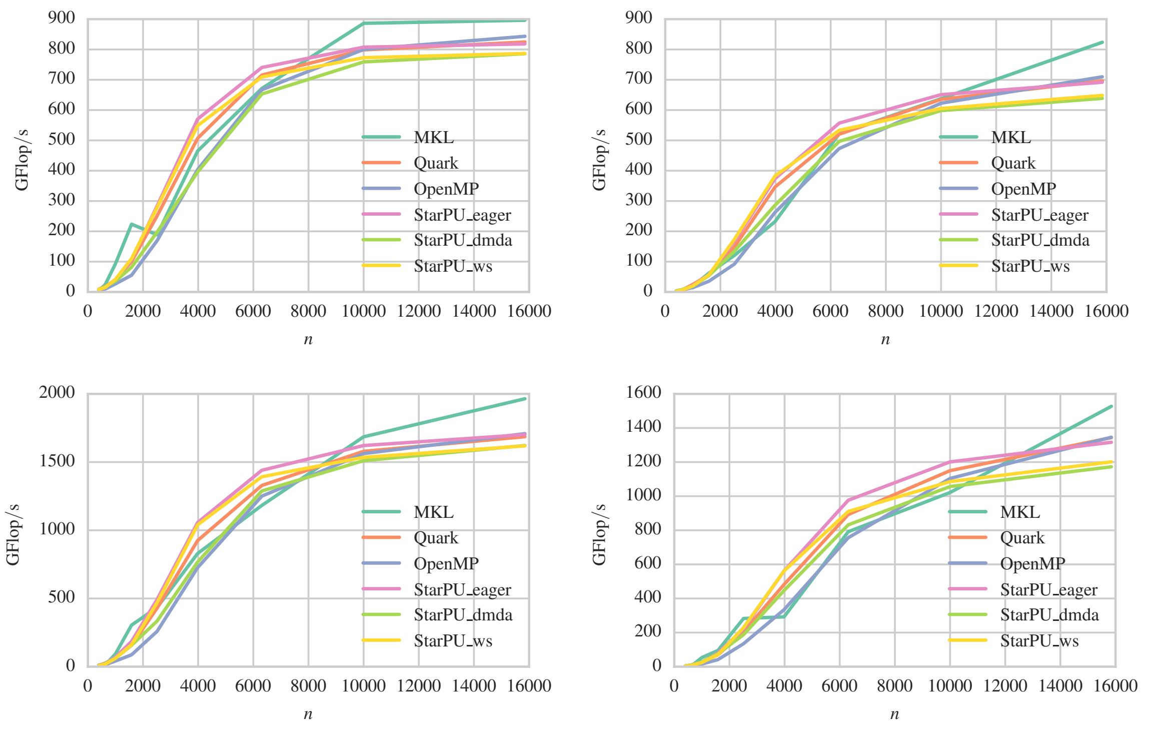


Figure 7: Performance of QR factorization on NUMA node. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, February 2011.

[6] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *CoRR*, abs/0709.1272, 2007.

[7] Jack J Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile lu factorization. 2011.

[8] John Gunnels and Robert van de Geijn. Developing linear algebra algorithms: A collection of class projects. FLAME Working Note #3. Technical Report TR-2001-19, The University of Texas at Austin, Department of Computer Sciences, May 2001.

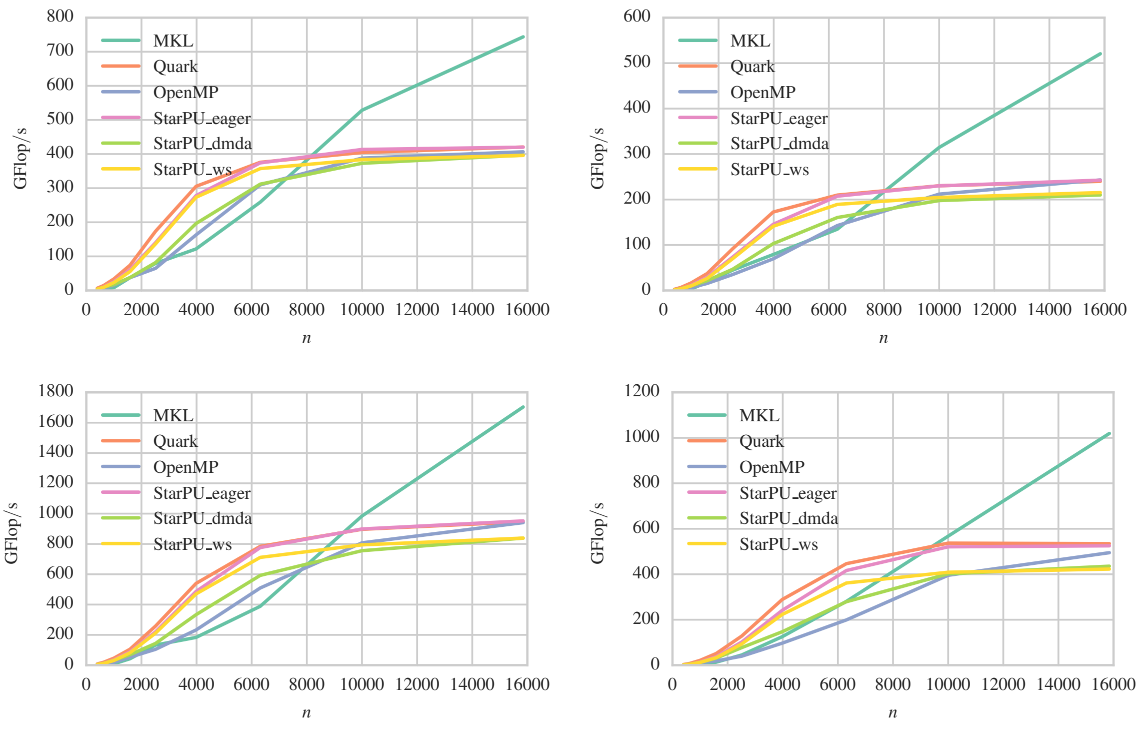


Figure 8: Performance of QR factorization on the KNL. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

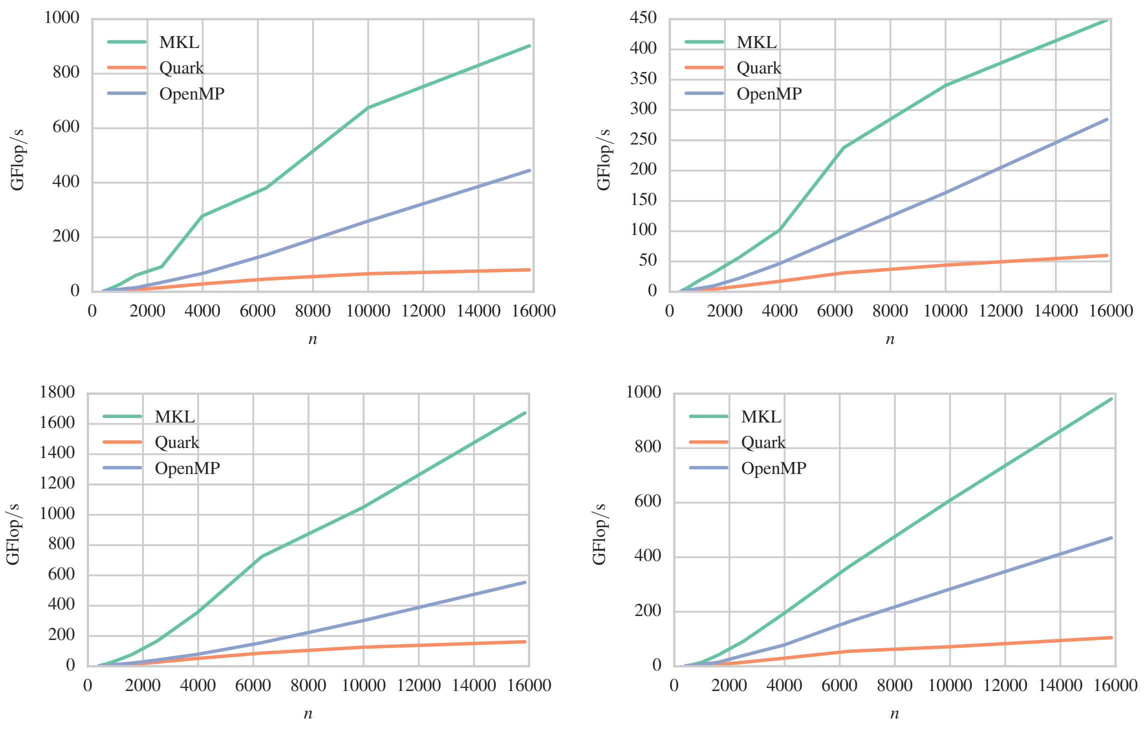


Figure 9: Performance of symmetric-indefinite factorization on NUMA node. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.

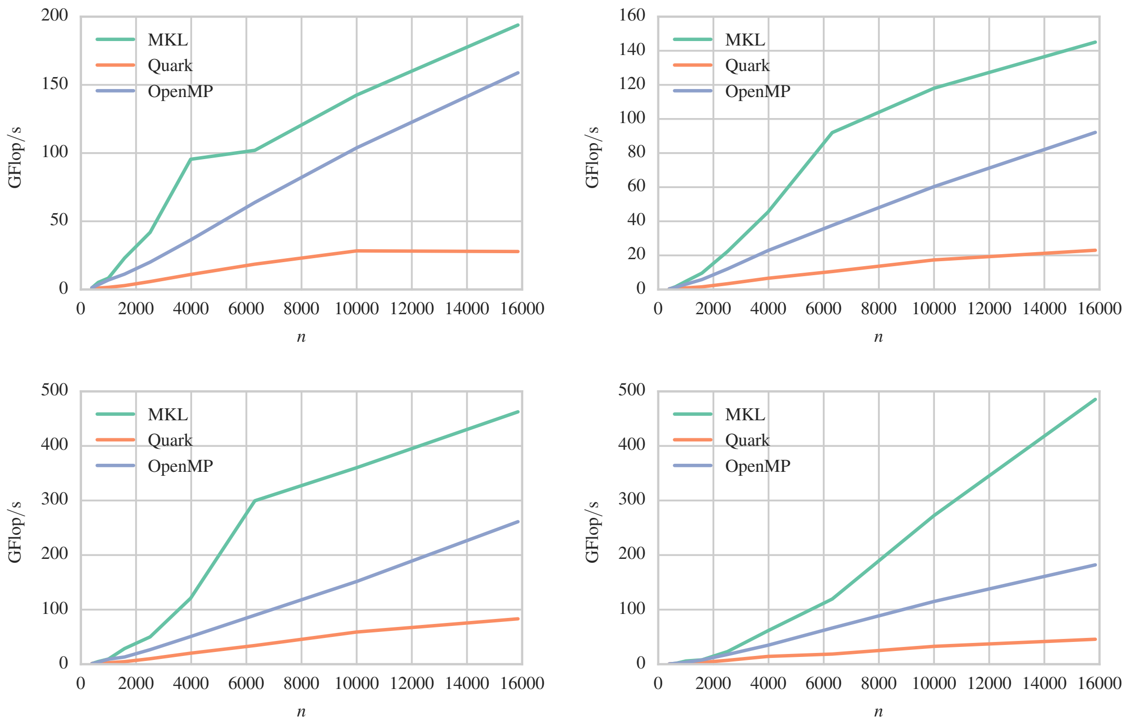


Figure 10: Performance of symmetric-indefinite factorization on the KNL. The top row has double complex precision on the left and double precision on the right. The bottom row has complex precision on the left and single precision on the right.